Assignment 6
Surfin' U.S.A.

by Jess Srinivas
edited by Ben Grant
Based on *The Perambulations of Denver Long* by Professor Darrell Long
CSE 13S, Winter 2024
Document version 1 (changes in Section 12)

Due Wednesday February 28th, 2024
Draft Due Monday February 26th, 2024

1 Introduction

As students of the best coastal university in the United States, if not the world, you have probably seen Santa Cruz's surfing culture. Santa Cruz is so well known for its surf culture that the city of Santa Cruz and surfing are almost synonymous. Santa Cruz is also one of the cities mentioned in the Beach Boys' 1963 song, *Surfin' U.S.A.*

Jess's friend, Alissa, wants to jump into the ocean at every city[1] mentioned in the song to prove that Santa Cruz has the best oceans and beaches. Because Alissa is a broke college student, she can't afford gas, and therefore must get between the cities on a budget (Figure 1). Alissa will be starting her trip in Santa Cruz on Friday morning (March 1st), and will spend some time in each beach. She does not want to repeat any beaches during her trip as she must return back to Santa Cruz so she can feed her roommate's cat, Tank at the end of the day.

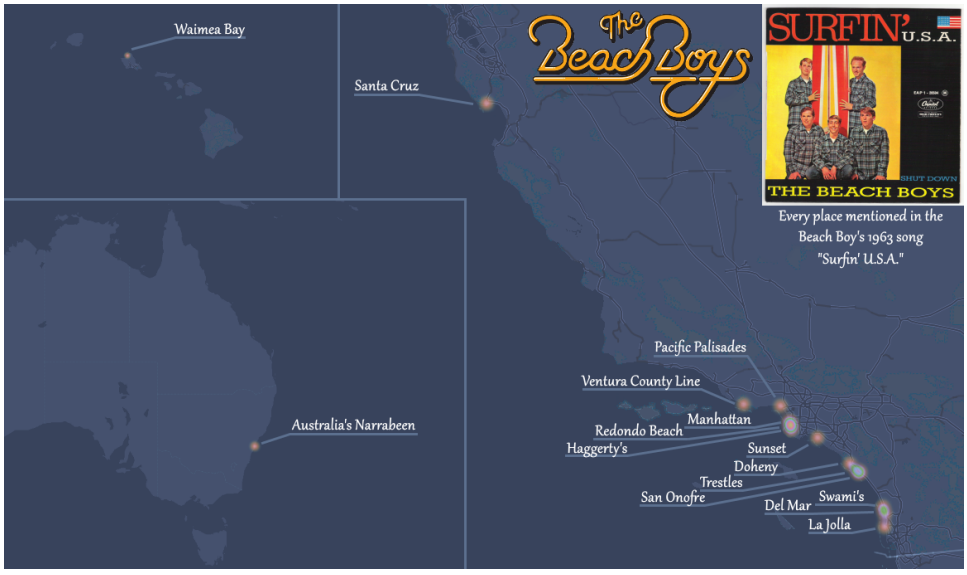


Figure 1: Every place mentioned in the Beach Boy's 1963 song "Surfin' U.S.A."[1]

[1]Because international travel is a pain, and it might be a bad idea for Alissa to drive her Prius into the ocean, Alissa will not be travelling to Narrabeen or Waimea Bay.

In this assignment, your task will be to use graph theory to help her get to each city using the smallest amount of gas. You must make sure that her route starts and ends in Santa Cruz, and she visits every other city exactly once. While you could in theory make a list of every possible route between every city, this is inefficient, and will very quickly create a lot of work for you. Instead, you will use graph theory, depth first search, and a stack to implement a solution to the classic Travelling Salesman Problem.

2 Graphs

2.1 General Info

A graph is a structure used frequently in discrete mathematics. There are two (main) types of graphs: a directed graph, and an undirected graph. A graph consists of two basic parts: vertices and edges. A vertex is a point in space: in this example, a city. An edge is a line connecting two vertices. This has a "weight". This can be thought of like the amount of time it takes to travel between two cities. In a undirected graph, the weight of the edge between two cities is the same, regardless of the direction. In a directed graph, the weight can be different based on direction, or the edge could only connect the vertices in one direction. This means that you can travel directly from point A to point B, but not from point B to point A.

Some other important features to note in a graph are cycles and loops. A loop is an edge from a vertex to itself. This can have any weight, not necessarily zero, but also does not need to exist. Also note the existence of a cycle. A cycle is a path that takes you back to where you started. You will read more about cycles when you read about the Travelling Salesman Problem (TSP, in Section 5). Finally, while negative edges are possible in a graph, you will not find them in this assignment.

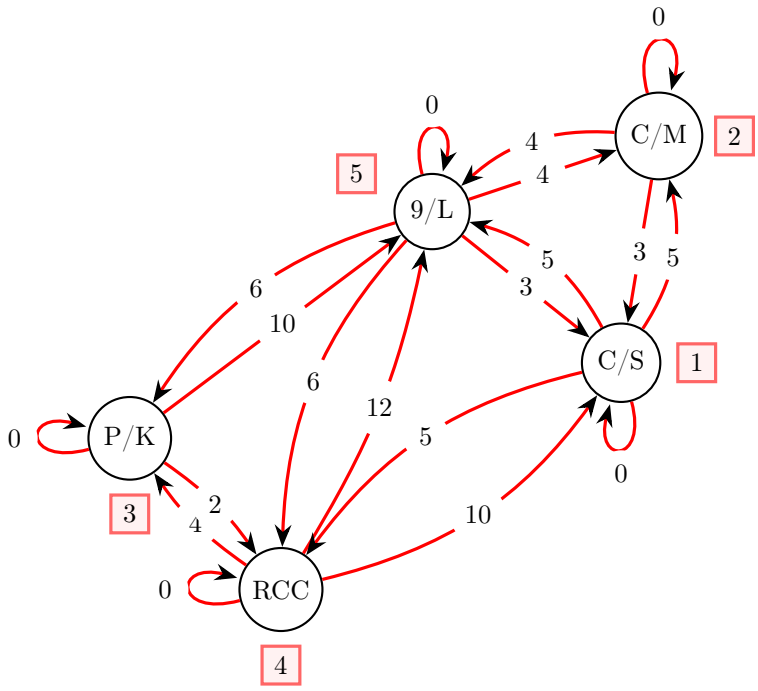


Figure 2: A graph of the difficulty to get food on campus[2].

This graph of the locations to get food on campus shows a lot of the features of graphs. On the other hand, if you are at Crown/Merrill, you can get food without going anywhere, and therefore taking zero effort (a loop). This is also clearly a directed graph, as it is harder to get from Porter/Kresge Dining Hall to College 9/Lewis as you are going uphill. Also notice that if you are travelling from Crown/Merrill, and your only goal is to get food, you might as well stop at 9/Lewis, as it's on the way. There is no path from Crown/Merrill to Porter/Kresge that doesn't pass by 9/Lewis.

This leads us to the question of: *how do we store the graph?* There are two ways that are usually used to store a graph. Both of these ways work to store a directed graph, but they also both have trade-offs.

From		To	Weight
C/S	→	C/S	0
C/S	→	C/M	5
C/S	→	9/L	5
C/M	→	C/S	3
	⋮		⋮
9/L	→	P/K	6
9/L	→	9/L	0

Table 1: An (incomplete) adjacency list.

The first is called an adjacency list. (See Table 1.) An adjacency list stores a graph as a list of pairs of edges, and a weight between them. This works fine for a graph with few edges, but is a lot harder with a graph that has many edges. The reason for this is that if we have to find an edge, we must search the whole list before we know if it exists. Take a note of how hard it would be to find an edge weight if you were given a complete adjacency list for this graph! An adjacently list is still useful to us if we need to store the list in a minimal amount of space, as it only stores edges that exist, and therefore no space is wasted.

	C/S	C/M	P/K	RCC	9/L
C/S	0	5	X	5	5
C/M	3	0	X	X	4
P/K	X	X	0	2	10
RCC	10	X	4	0	12
9/L	3	4	6	6	0

Figure 3: The adjacency matrix for the dining hall graph[2].

The other way to store a graph is using an adjacency matrix. (See Figure 3.) If we have a graph with E edges, our adjacency matrix is an $E \times E$ matrix, where for every a in $(0, 1, 2, \ldots, E-1)$ and for every b in $(0, 1, 2, \ldots, E-1)$, M_{ab} is the weight of the edge between Vertex A and Vertex B

2.2 Assignment Specifics

In the case of Alissa's tour of California, you should treat all edges as undirected unless told otherwise (you handle undirected graphs by adding a second edge in the opposite direction). Obviously, the edges will also have weights. You will be storing your weights in an **adjacency matrix**. You will also store an extra array of strings holding the names of the vertices on the graph.

Here is the completed struct, as defined in `graph.c`

```
typedef struct graph {
    uint32_t vertices;
    bool directed;
    bool *visited;
    char **names;
    uint32_t **weights;
} Graph;
```

Since this struct is only defined in `graph.c`, files that use the graph (which need to include `graph.h`) only know that some struct called `Graph` exists; they do not know what its members are. This means the

only way they can manipulate a Graph is by storing a pointer to it and calling the functions declared in `graph.h`.

2.3 Functions

Your graph will have the following functions:

`Graph *graph_create(uint32_t vertices, bool directed)`

Creates a new graph struct, and returns a pointer to it. Initializes all items in the `visited` array to `false`.

```
Graph *graph_create (uint32_t vertices, bool directed) {
    Graph *g = calloc(1, sizeof(Graph));
    g->vertices = vertices;
    g->directed = directed;
    // use calloc to initialize everything with zeroes
    g->visited = calloc(vertices, sizeof(bool));
    g->names = calloc(vertices, sizeof(char *));
    // allocate g->weights with a pointer for each row
    g->weights = calloc(vertices, sizeof(g->weights[0]));

    // allocate each row in the adjacency matrix
    for (uint32_t i = 0; i < vertices; ++i) {
        g->weights[i] = calloc(vertices, sizeof(g->weights[0][0]));
    }

    return g;
}
```

`void graph_free(Graph **gp)`

Frees *all* memory used by the graph. Take a close look at when memory is allocated in this function to be sure you free everything! Double check with `valgrind`.

This function takes in a double pointer (`Graph **`; a pointer to a pointer) so that it can set the Graph pointer (that `gp` points to) to `NULL`. If it didn't do this, it would be possible to try to access the Graph using a pointer that had already been freed. Using a value after freeing it has the potential to create serious vulnerabilities. To avoid this, we set `*gp` to `NULL` to ensure that any attempt to use the Graph after freeing it will immediately crash your program.

`uint32_t graph_vertices(const Graph *g)`

Finds the number of vertices in a graph.

`void graph_add_vertex(Graph *g, const char *name, uint32_t v)`

Gives the city at vertex v the name passed in. This function makes a copy of the name and stores it in the graph object. The parameter is `const` to indicate that `graph_add_vertex()` cannot modify the string passed into it.

```
if (g->names[v]) free(g->names[v]);
g->names[v] = strdup(name);
```

The `strdup()` function makes a copy of the string, which is necessary (and means it must be freed later). Otherwise, you wouldn't be sure that the pointer passed into `graph_add_vertex()` would still be valid as long as the Graph existed (for instance, the caller might pass in a buffer which they will later use for something else). We also make sure that if we overwrite an existing name, the old one is freed.

`const char* graph_get_vertex_name(const Graph *g, uint32_t v)`

Gets the name of the city with vertex v from the array of city names. This does not allocate a new string, it simply returns the one stored in the Graph. `g` is `const` since this function doesn't need to modify the Graph, and its return type is `const` to prevent the caller from manipulating the string that it returns.

`char **graph_get_names(const Graph *g)`

Gets the names of the every city in an array. Returns a double pointer – an array of strings – but *not a copy.*

`void graph_add_edge(Graph *g, uint32_t start, uint32_t end, uint32_t weight)`

Adds an edge between `start` and `end` with weight `weight` to the adjacency matrix of the graph.

`uint32_t graph_get_weight(const Graph *g, uint32_t start, uint32_t end)`

Looks up the weight of the edge between `start` and `end` and returns it.

`void graph_visit_vertex(Graph *g, uint32_t v)`

Adds the vertex `v` to the list of visited vertices.

`void graph_unvisit_vertex(Graph *g, uint32_t v)`

Removes the vertex `v` from the list of visited vertices.

`bool graph_visited(const Graph *g, uint32_t v)`

Returns `true` if vertex `v` is visited in graph `g`, `false` otherwise.

`void graph_print(const Graph *g)`

Optionally, prints a human-readable representation of a graph. Even though this function is not required, we *strongly recommend* that you implement it as it will aid in making sure that you are reading in the graph correctly.

2.4 The `.graph` format

You will write a function that reads a text-file representation of a graph, creates a new Graph, and returns a pointer to it. The text file has a name that ends in `.graph`. The `.graph` file contains several lines, grouped into four sections, which are defined in Table 2 below. Each line has a maximum length of `PATH_MAX` as defined in `<limits.h>`

Another example graph, `basic.graph` is as follows (notated with C style comments)

```
2           // There are 2 vertices in the graph
Home        // The first vertex is called "home"
The Beach   // The 2nd vertex is called "The Beach"
2           // There are two edges in this Graph
0 1 1       // The first edge is from 0 (home) to 1 (the beach) with a weight of 1
1 0 2       // The other edge is from 1 back to 0 with a weight of 2
```

For a more detailed example, you can look at any other `.graph` file provided in the resources repository.

Section	Purpose	Example
1	One line: Number of Vertices	`3`

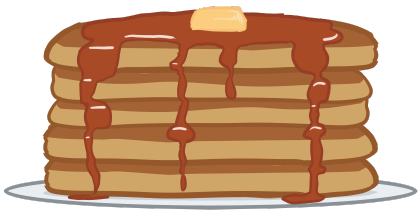
2	Several lines: Names of Vertices. If the number of vertices is n, there are n names, one name per line.	`Home`
		`Beach`
		`Bookstore`

3	One line: Number of Edges	`2`
4	Several lines: All edges and weights as a adjacency list, in the format `Start End Weight`, ex: `0 1 5` would represent the edge from `Home` to `Beach` (with a weight of 5). The numbers correspond to the vertices listed in Section 2, with the first vertex being number `0`. There are exactly as many edges in this list as specified in the line above.	`0 1 5` `1 2 7`

Table 2: Format of a `.graph` file.

3 Stacks

3.1 General Information



[3]

A stack is an abstract data type used to store a list of elements. Unlike arrays, a stack only has two essential operations: Push, and Pop. This means that it does not have random access; that is to say that you can not access an element in the middle of the stack in any way. It is a good idea to think of a stack like you would think about a stack of pancakes. When you create a new pancake, you must add it to the top of the stack. When somebody grabs a pancake to eat, they must remove it from the top of the stack. This means that a stack follows a last in - first out (LIFO) ordering. You can't remove a pancake from the middle of the stack.

3.2 Assignment Specifics

Our implementation of a stack will be used to track the path that Alissa takes on her trip. We will use an array (`items`)to represent the list of elements. Because we modify the last element of the array, we need a variable to track where the end of the stack resides in your computer's memory. This will be `stack->top`. In order to not wastefully allocate memory, your path ADT will dynamically allocate the array inside. It will only do this once, and it will store the maximum capaity of the stack in the struct field `struct->capcacity`.

Here is the complete struct, as defined in `stack.c`.

```
typedef struct stack {
    uint32_t capacity;
    uint32_t top;
    uint32_t *items;
} Stack;
```

3.3 Functions

The stack that you implement will have push and pop, along with a few functions to make them easier for you to use:

`Stack *stack_create(uint32_t capacity)`

Creates a stack, dynamically allocates space for it, and returns a pointer to it.

```
// Attempt to allocate memory for a stack
// Cast it to a stack pointer too!
Stack *s = (Stack *) malloc(sizeof(Stack));
s->capacity = capacity;
s->top = 0;
// We need enough memory for <capacity> numbers
s->items = calloc(s->capacity, sizeof(uint32_t));
// We created our stack, return it!
return s;
```

While `malloc` and `calloc` can return a null pointer, we will assume that this will not happen during this assignment. If you want, you should get in the habit of checking for that edge case when you allocate memory.

Also note that the value of `stack->top` is 0 on an empty list. This does not mean that there is an item at index 0. The `stack->top` field simply points to the next *empty* slot on the stack.

`void stack_free(Stack **sp)`

Frees all space used by a given stack, and sets the pointer to `NULL`.

```
// sp is a double pointer, so we have to check if it,
// or the pointer it points to is null.
if (sp != NULL && *sp != NULL) {
    // Of course, we have to remember to free the
    // memory for the array of items first,
    // as that was also dynamically allocated!
    // If we freed the Stack first then we would
    // not be able to access the array to free it.
    if ((*sp)->items) {
        free((*sp)->items);
        (*sp)->items = NULL;
    }
    // Free memory allocated for the stack
    free(*sp);
}
if (sp != NULL) {
    // Set the pointer to null! This ensures we dont ever do a double free!
    *sp = NULL;
}
```

`bool stack_push(Stack *s, uint32_t val)`

Adds `val` to the top of the stack S, and increments the counter. Returns `true` if successful, `false` otherwise (ex: the stack is full).

```
    // If the stack is full, return false;
    if (stack_full(s)) {
        return false;
    }
    // Set val
    s->items[s->top] = val;
    // Move the top of the stack
    s->top++;
    return true;
```

`bool stack_pop(Stack *s, uint32_t *val)`

Sets the integer pointed to by val to the last item on the stack, and removes the last item on the stack. Returns `true` if successful, `false` otherwise. Remember that `stack->top` is **not** the index of the top value of the stack.

`bool stack_peek(const Stack *s, uint32_t *val)`

Sets the integer pointed to by val to the last item on the stack, but *does not modify the stack*. Returns `true` if successful, `false` otherwise. Remember that `stack->top` is **not** the index of the top value of the stack.

`bool stack_empty(const Stack *s)`

Returns `true` if the stack is empty, `false` otherwise. A stack is empty when there are no elements in it.

`bool stack_full(const Stack *s)`

Returns `true` if the stack is full, `false` otherwise. A stack is full when the number of elements is equal to the capacity.

`uint32_t stack_size(const Stack *s)`

Returns the number of elements in the stack. An empty stack contains zero elements.

`void stack_copy(Stack *dst, const Stack *src)`

Overwrites `dst` with all the items from `src`. You should also make sure to update `dst->top` so that your code knows how many items are now in the stack. Finally, although it's unlikely to come up in this assignment as all your Stacks will be the same length, we should consider that `dst` may not have a capacity large enough to store every item from `src`. You can use the `assert` function to make sure that this is not the case (if it does happen, it is likely an error in your code).

`void stack_print(const Stack* s, FILE *outfile, char *cities[])`

This function will print out the stack as a list of elements, given a list of vertex names, starting with the *bottom* of the stack. Every item in the stack should be less than the number of vertices so you can index the array successfully.

```
    for (uint32_t i = 0; i < s->top; i += 1) {
        fprintf(outfile, "%s\n", cities[s->items[i]]);
    }
```

4 Paths

In this assignment, a path will be the data structure we use to track Alissa's travels. Because we are tracking her route, we need to keep track of the distance travelled in order to make sure that we store the shortest path. A path will use the following struct:

```
typedef struct path {
    uint32_t total_weight;
    Stack *vertices;
} Path;
```

The Path ADT will require the following functions:

4.1 Functions

`Path *path_create(uint32_t capacity)`

Creates a path data structure, containing a Stack and a weight of zero.

`void path_free(Path **pp)`

Frees a path, and all its associated memory.

`uint32_t path_vertices(const Path *p)`

Finds the number of vertices in a path.

`uint32_t path_distance(const Path *p)`

Finds the distance covered by a path.

`void path_add(Path *p, uint32_t val, const Graph *g)`

Adds vertex `val` from graph `g` to the path. This function must also update the distance and length of the path. When adding a vertex to an empty path, the distance should remain zero. Otherwise, you must look up the distance from the most recent vertex to the new one and add that to the total weight. The distance can be non-zero only when there are at least two cities in the path.

`uint32_t path_remove(Path *p, const Graph *g)`

Removes the most recently added vertex from the path. This function must also update the distance and length of the path based on the adjacency matrix in the graph pointed to by `g`. When removing the last vertex from a path, the distance should become zero. The distance can be non-zero only when there are at least two cities in the path.

Since the return value of this function was not specified in the original version of this assignment, we will not require you to return anything specific. However, you'll probably find that returning the index of the removed vertex is the most useful for your DFS implementation.

`void path_copy(Path *dst, const Path *src)`

Copies a path from `src` to `dst`.

`void path_print(const Path *p, FILE *outfile, const Graph *g)`

Prints the path stored, using the vertex names from `g`, to the file `outfile`. See the Section 8 for the exact form of the print statement. This function should only print the names of the vertices. The rest of the output is produced by `tsp.c`.

5 The Travelling Salesman Problem

5.1 General Info

The Travelling Salesman Problem is a classic graph theory problem with a simple premise: what is the shortest path on a graph that starts and ends at the same point, visiting every vertex exactly once. Let's break that down...

- a path is just any set of vertices where the first is connected to the 2nd, and the 2nd is connected to the third, etc.
- A path that starts and ends at the same point is called a cycle. A cycle can be of any length, including 1. A cycle can also visit a vertex more than once.
- A cycle that visits every vertex exactly once is called a Hamiltonian cycle.
- The Hamiltonian cycle with the smallest weight is the solution to the travelling salesman problem.

How does one go about solving the travelling salesman problem? On a smaller set, you can just use brute force. If you make a map of every single possible path that you can take, you will probably encounter the solution. The only issue there is that you may encounter the same vertex multiple times, or get stuck at one vertex with no way out.

To get out of this, we can use a graph search algorithm. The two most important ones are depth first search (DFS) and breadth first search (BFS).

The pseudocode for `dfs` is as follows:

```
def dfs(node n, graph g):
    mark n as visited
    for every one of n's edges (n, n2):
        if (n2 is not visited):
            dfs(n2, g)
    mark n as unvisited
```

This algorithm will eventually traverse every single possible path.

5.2 Assignment Specifics

Knowing every single path might help Alissa, but not all of them will be able to get her to every beach, or get her home. Once we have found a path, we should first check if it's a path that Alissa can take. After all, Jessie would be quite sad if her friend left to go to the beach, never to return. If we know that every beach has only been visited once, and we know how many beaches were on the list of places to visit, we can simply check if the number of beaches that she has visited is the same as the number of beaches that were on the list.

Then, we need to make sure she finds a way home. We can do this by checking if the beach she ends at has a path back to Santa Cruz. If that path exists, we then have to see if its the most fuel efficient path. We can do this by storing our current best path until we find a better alternative[2], and comparing them. The path at the end will be the most efficient path!

Although it doesn't technically make a difference as the paths we find are loops, your paths should start and end at the vertex `START_VERTEX`, defined in `vertices.h`. It may be the case that multiple Hamiltonian cycles with the same shortest length exist. if this is the case, it's okay for your code to find any of them, not just the same one that the resources binary finds.

[2]We must find a better alternative, meaning that we take the first best path we find if there is a tie.

6 File IO

6.1 General Info

File IO is short for "File input and output". So far in this course, you have only read in from the command line arguments or using `scanf`. While these are useful tools and good ways to read data into your program, we will need to begin reading and writing to files on your computer. Without knowing it, you have actually been reading and writing to files this whole course. The `printf()` function in C writes to a special file called `stdout`, and the `scanf()` function reads from `stdin`. These "files" are defined in `stdio.h`. By default, `stdout`, `stderr` (an alternative stream of output often used for errors), and `stdin` are connected to your terminal window, but as you may know you can redirect and pipe them as well. In this assignment, you will read and write to files using `fscanf` and `fprintf` respectively. The "f" before `scanf` and `printf` is short for "file". The `fprintf` and `fscanf` functions add one more argument, which is a file pointer. The functions will output the same text to the file referred to by the file pointer.

To get a file pointer for any file that isn't `stdin` or `stdout` use the `fopen()` function. For example, to open a file for reading, use C program lines like these:

```
FILE *infile = stdin;
if (dash_i) {
    infile = fopen(input_filename, "r");
    // TODO confirm that infile is not NULL
}
```

Then call `fscanf()` to read data from the input file. For example, here is how to read a decimal number from an input file into a `uint32_t` variable `num_vertices`:

```
if (fscanf(infile, "%u\n", &num_vertices) != 1) {
    fprintf(stderr, "tsp:  error reading number of vertices\n");
    exit(1);
}
```

Finally, once you are done using the file, you must also use the `fclose()` function to close the file and free all associated memory. (However, you don't need to call `fclose()` on `stdin` or `stdout`.)

```
if (infile != stdin) fclose(infile);
```

The use of `fopen()` with an output file is similar, except that the second parameter of `fopen()` is `"w"`, and instead of using `fscanf()` you use `fprintf()`.

6.2 Assignment Specifics

In this assignment, you will have to read in from text files that contain representations of graphs. Here is a checklist of reminders for when you use `fopen()`, `fscanf()`, and `fprintf()`.

- ☐ Check the return value from `fopen()`. The function returns `NULL` when it can't open a file. If `fopen()` returns `NULL`, report an error, and of course don't use the return value.
- ☐ Check the return value from `fscanf()`. The function returns the number of items successfully read. If there is one "`%`" in `fscan()`'s format string, then it returns 1 for success. For example, the function call `fscanf(f, "%u\n", &n)` should return 1. If "`%`" appears three times in its format string, then `fscanf()` returns 3 for success. For example, `fscanf(f, "%u%u%u", &i, &j, &k)` should return 3.
- ☐ Use `"r"` as the second parameter of `fopen()` when you want to read an existing file using `fscanf()`. For example: `infile = fopen(input_file_name, "r")`.
- ☐ Use `"w"` as the second parameter of `fopen()` when you want to write a new file using `fprintf()`. For example: `outfile = fopen(output_file_name, "w")`.
- ☐ Use a default value for the file pointers. `FILE *infile = stdin; FILE *outfile = stdout;`. Then overwrite the file pointers if either the `-i` or `-o` option is specified (see Section 7.2).

7 Command line options

7.1 General info

Command line options are very useful to bring data into a program. Be sure to read Ben's guide to command line options[4] so you can implement them with ease in this assignment and in the future.

7.2 Assignment Specifics

- `-i` : Sets the file to read from (input file). Requires a filename as an argument. The default file to read from is `stdin`
- `-o` : Sets the file to write to (output file). Requires a filename as an argument. The default file to write to is `stdout`
- `-d` : Treats all graphs as *directed*. Remember that the default is to assume an *undirected* graph, which means that any edge (i, j) that is specified should be added as both (i, j) and (j, i). So if `-d` is specified, then (i, j) will be added, but (j, i) *won't*.
- `-h` : Prints a help message to `stdio`.

8 Program Output and Error Handling

If any invalid options or files are specified, your program should exit cleanly, and print the help message and an error message to `stderr`. See File IO in Section 6 to learn how to do that. This is **not** the same as in previous assignments, where you only had to print to `stdout`.

The output of `./tsp -d -i maps/basic.graph` will be as follows:

```
Alissa starts at:
Home
The beach
Home
Total Distance: 3
```

The output of `./tsp -d -i maps/lost.graph` (and any other graph with no valid path) will be as follows:

```
No path found! Alissa is lost!
```

The words "No path found! Alissa is lost!" will always be printed for any graph with no Hamiltonian Cycles

You can assume that every graph will follow the format specified in Section 2.4 and will only contain positive integers for weights.

If some reason, an edge is duplicated, only count the last one. This could include something like running `./tsp -i maps/basic.graph`.

9 Testing your code

9.1 General Info

Part of the process of writing good code is being able to test it. Because of this, we expect that you write tests for each function you write, ensuring that they all function correctly and without errors. Testing individual functions is the best way to make sure that your code is fully functional before you integrate all the parts.

9.2 Assignment Specifics

While we will not check if you have written any tests for your code, we recommend that you create some test files, and test all parts of your code individually. Not doing so will make your assignment take far longer than you expect. We expect you to write your own test cases and describe them in your design.pdf

- You will receive a folder of graphs. Some of these graphs may contain unexpected bugs, features, or terrain. You will test your tsp.c against all of these graphs, and make sure that you are getting accurate results.
- Some of the functions that you are required to write may produce an error. You are required to handle these errors as defined in their specifications.
- Your program should have no *memory leaks*. Make sure you `free()` before exiting. `valgrind` should pass cleanly with any combination of the specified command-line options, including on an error condition.
- Your program must pass the static analyzer, `scan-build`, with no errors. Make sure you learn how to use it.

10 Submission

For the **design draft**, you must submit a commit ID on canvas *before Monday February 26th*. You must have a PDF called `design.pdf`.

For the **final version**, you must submit a commit ID with all required files to canvas before Wednesday February 28th. Not doing so will result in using late days.

1. Your program *must* have your source files.
   - `tsp.c` contains `main()` and *may* contain any other functions necessary to complete the assignment.
   - `graph.c` will contain the implementations for the graph ADT defined in `graph.h`
   - `stack.c` will contain the implementations for the stack ADT defined in `stack.h`
   - `path.c` will contain the implementations for the path ADT defined in `path.h`
2. Your submission also *must* include these files, which we have provided:
   You are allowed to add to these files as you wish, but you must implement all functions that are included in them, unchanged.
   - `graph.h` specifies the interface to `graph.c`.
   - `stack.h` specifies the interface to `stack.c`.
   - `path.h` specifies the interface to `path.c`.
   - `vertices.h` specifies which vertex the path starts at. See Section 5.2 for more details.
   - `Makefile` allows you to build your program. You may not modify this when you make your final submission (but you are free to change it while you test and debug)
3. You must also submit an updated version of your `design.pdf` that follows the template of previous assignments and answers the questions in Section 11. This should be completed before you start your code.

You can have other source and header files, but *do not try to be overly clever*. The header files for each of the helper functions are provided to you and may not be modified.

Properly submit your assignment through `git`. Remember: *add*, *commit*, and *push*, especially if you think the power may go out!

Your assignment is turned in *only* after you have pushed and submitted the commit ID you want graded on Canvas. "I forgot to push" and "I forgot to submit my commit ID" are not valid excuses. It is *highly* recommended to commit and push your changes *often*.

Notes:

1. Your code must be formatted using `clang-format` with the style provided in your repository. Running `make format` should achieve this.
2. Your code must compile successfully using your own `Makefile` (which must use the specified compiler flags).
3. You may not modify any header files.
4. You should clone the resources repository, which contains all required files, and use `cp` to move files to your personal repository.
5. Do not push any object files or binaries. You can use `make clean` before you push to ensure this.
6. Your code must pass `valgrind` on a variety of inputs and `scan-build` with no errors, leaks or warnings.

11 Questions To Be Answered In Your Design Plan

Along with the standard questions in your old design templates, answer the following questions:

- What benefits do adjacency lists have? What about adjacency matrices?
- Which one will you use. Why did we chose that (hint: you use both)
- If we have found a valid path, do we have to keep looking? Why or why not?
- If we find 2 paths with the same weights, which one do we choose?
- Is the path that is chosen deterministic? Why or why not?
- What type of graph does this assignment use? Describe it as best as you can
- What constraints do the edge weights have (think about this one in context of Alissa)? How could we optimize our dfs further using some of the constraints we have?

12 Revisions

Version 1 Original.

13 Pictures of Alissa's cat

Figure 4: Tank is such a good boy. Please make sure he gets fed.

Figure 5: Aww come on look at that cute face

References

[1] /u/Cryb111111111. Every place mentioned in the beach boys' song surfin' u.s.a, Sep 2021. URL: https://www.reddit.com/r/MapPorn/comments/pnvnmn/every_place_mentioned_in_the_beach_boys_surfin_usa/.

[2] Natalie Marks. Latex graph and adjacency matrix for finding food, May 2023.

[3] Blue Nguyen. a stack of pancakes, drawn by a former cse13s student, May 2023.

[4] Ben Grant. Options and arguments, April 2023. URL: https://13s-docs.jessie.id/c/options.html.